

# An Improved Interface to the CPN Simulator

**Lars M. Kristensen**  
**Michael Westergaard**

**Department of Computer Science**  
**University of Aarhus**



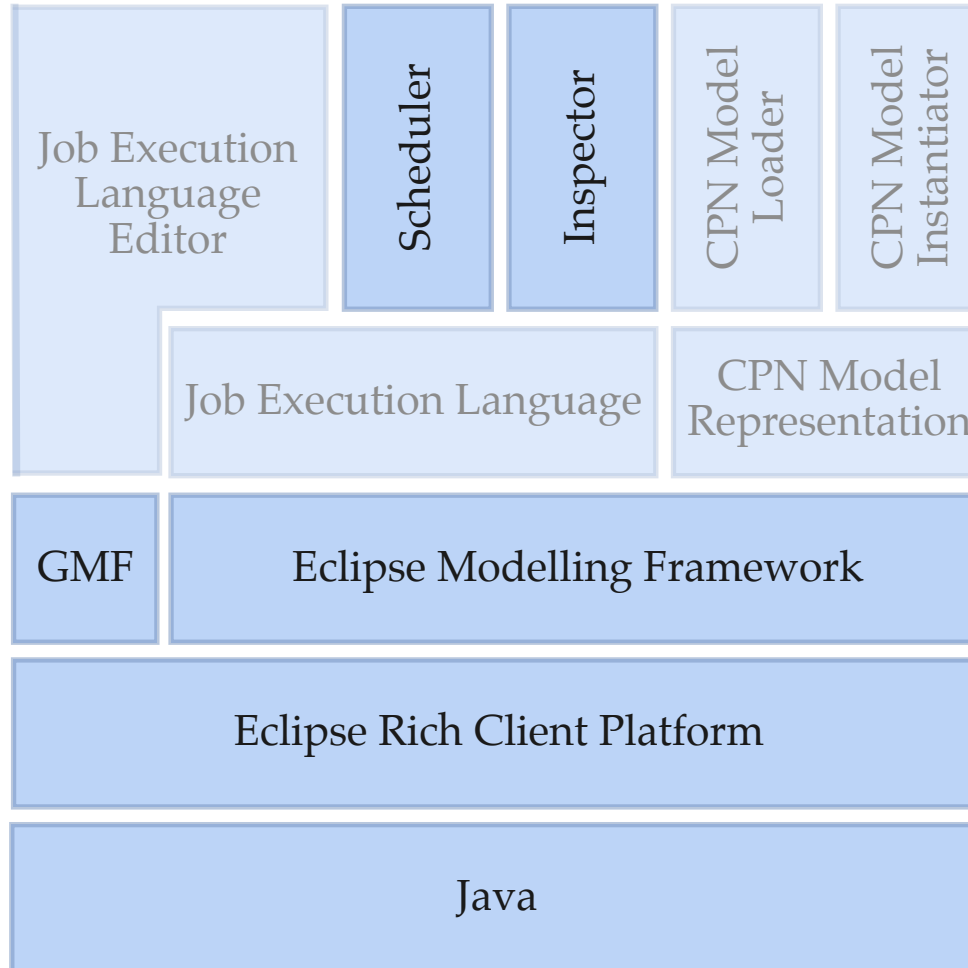
UNIVERSITY OF AARHUS

---

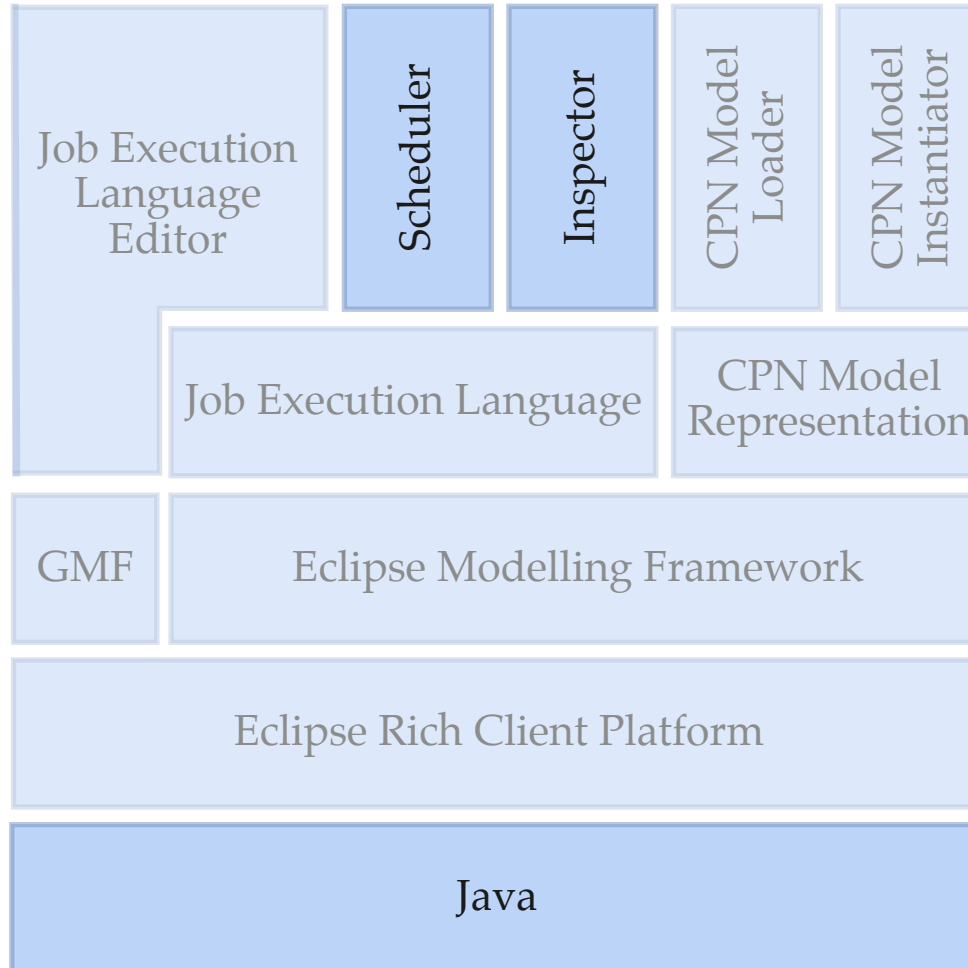
Department of Computer Science

Michael Westergaard

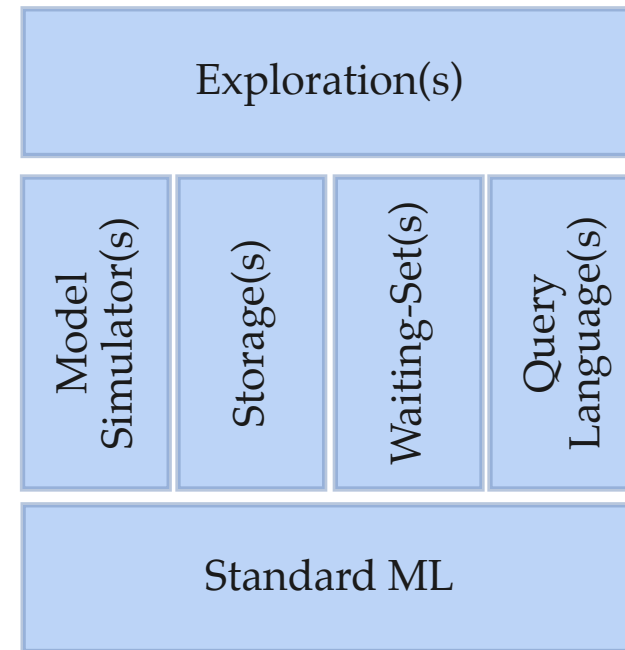
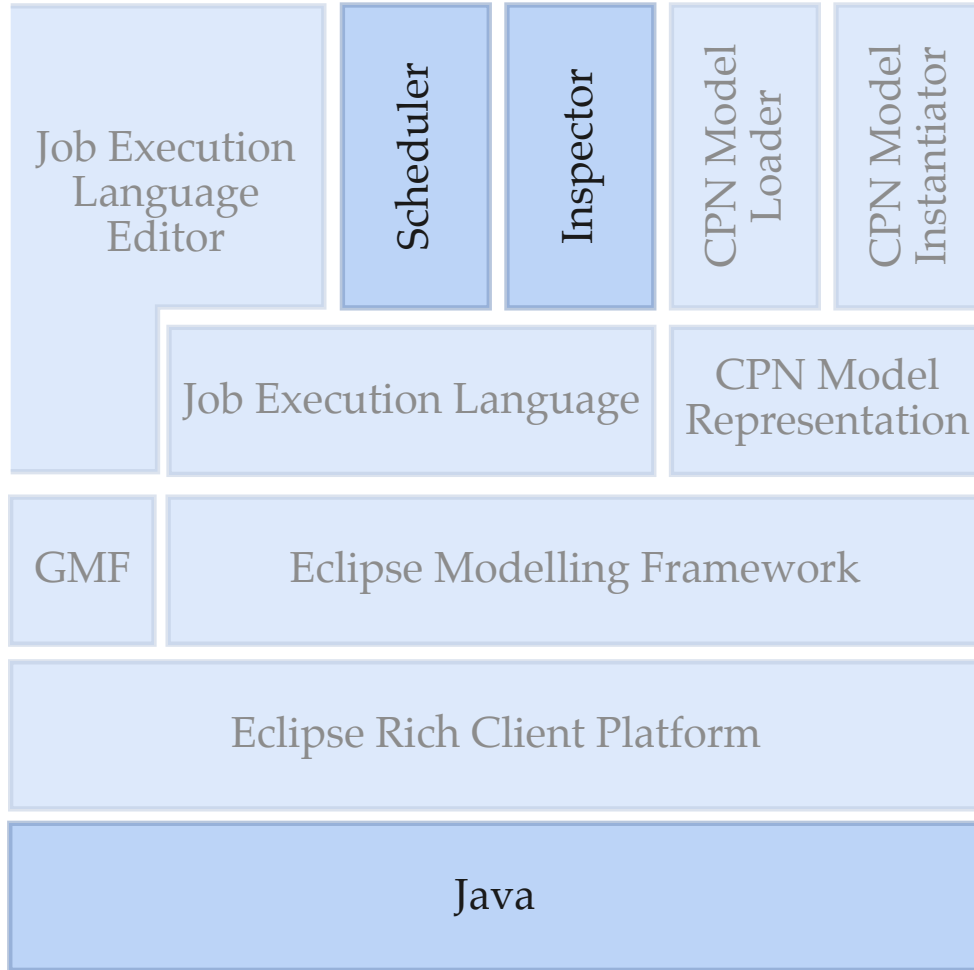
# Architecture



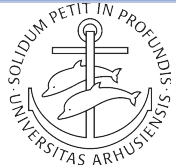
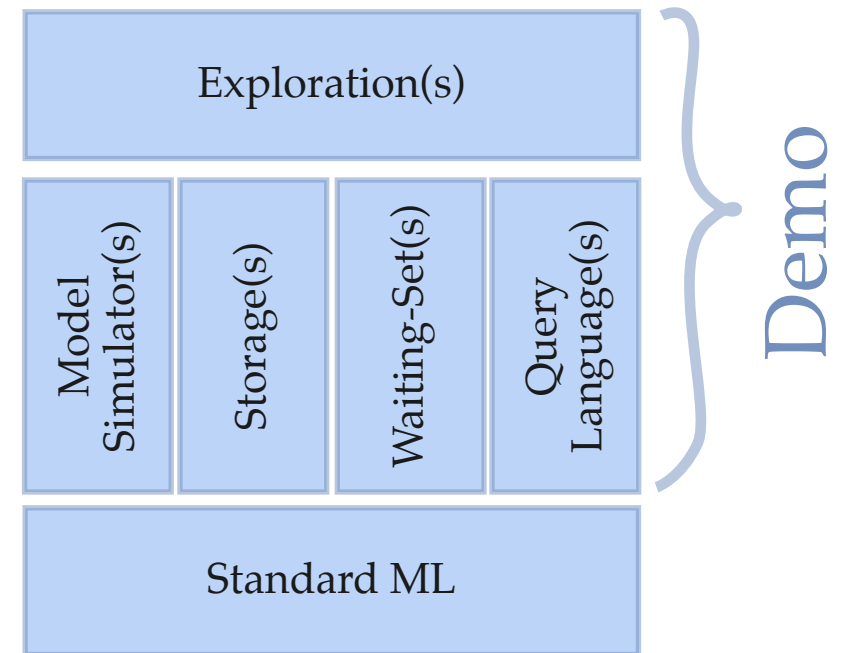
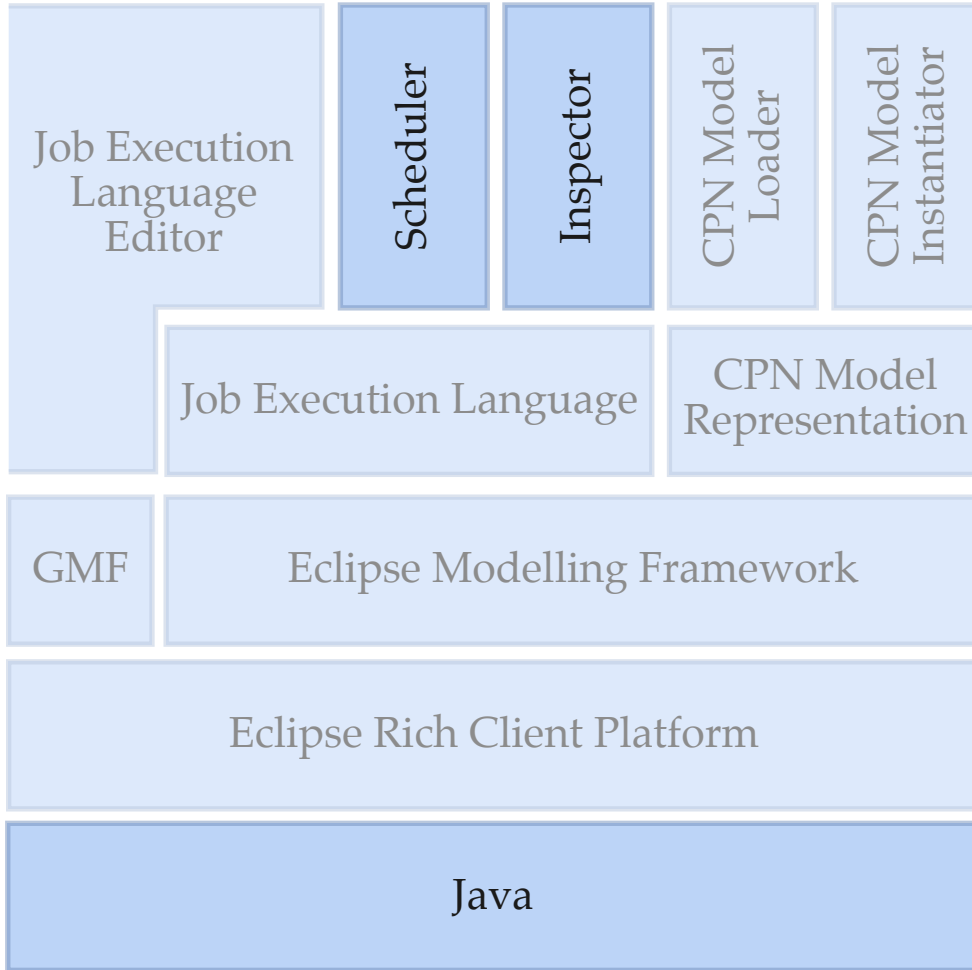
# Architecture



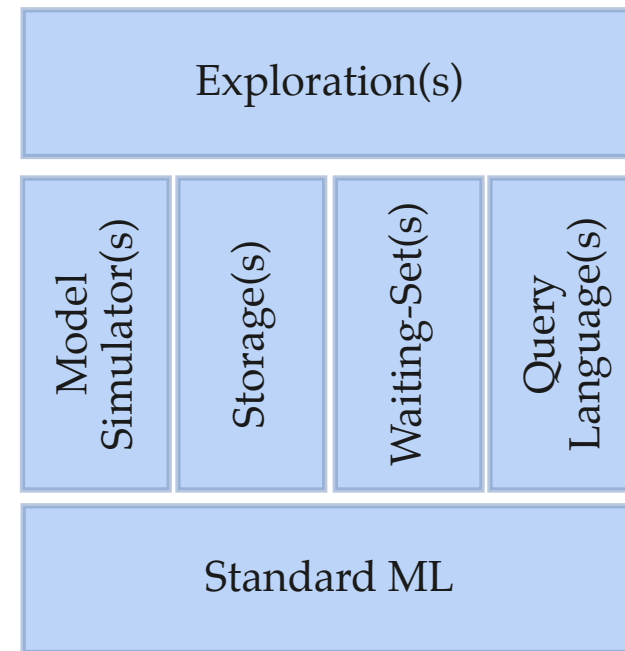
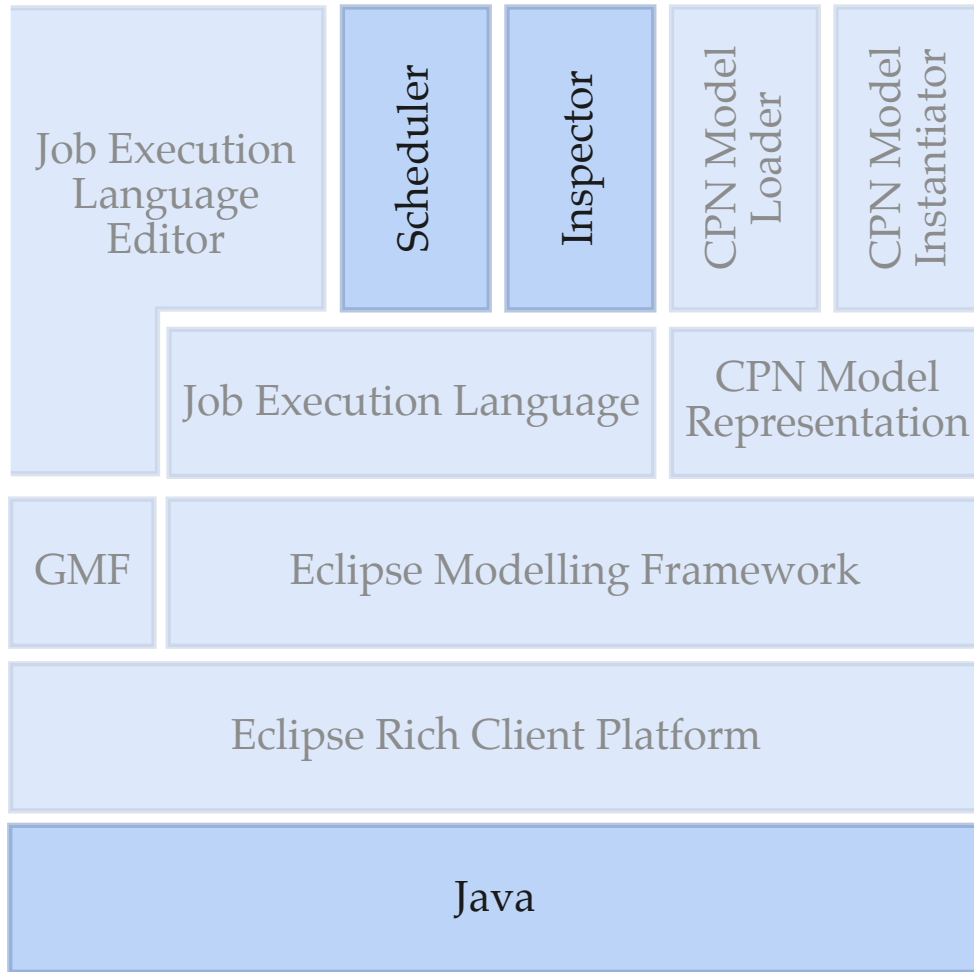
# Architecture



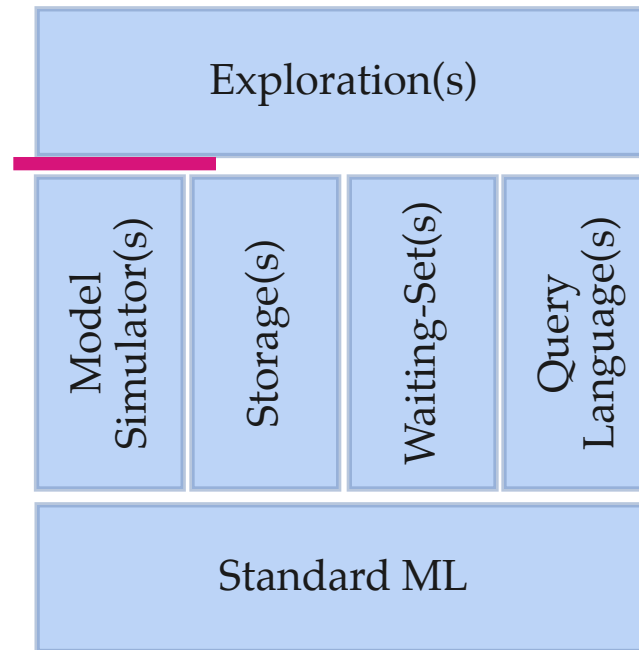
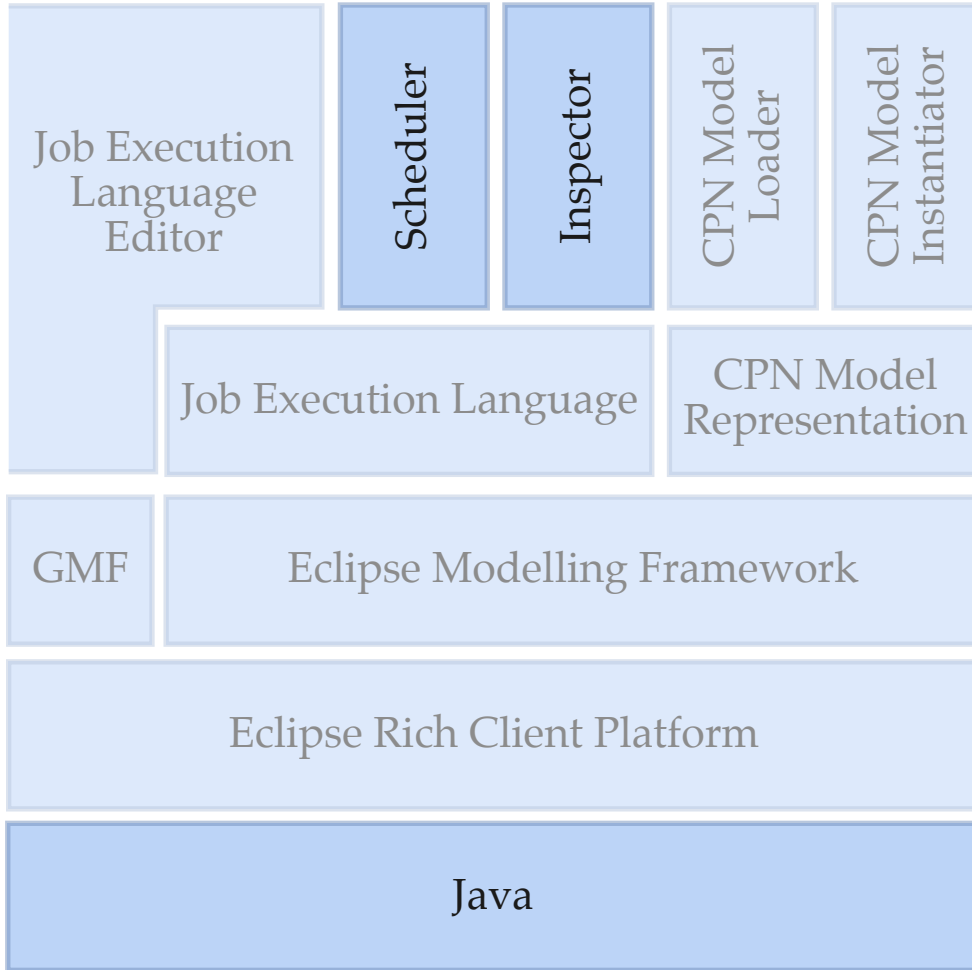
# Architecture



# Architecture



# Architecture

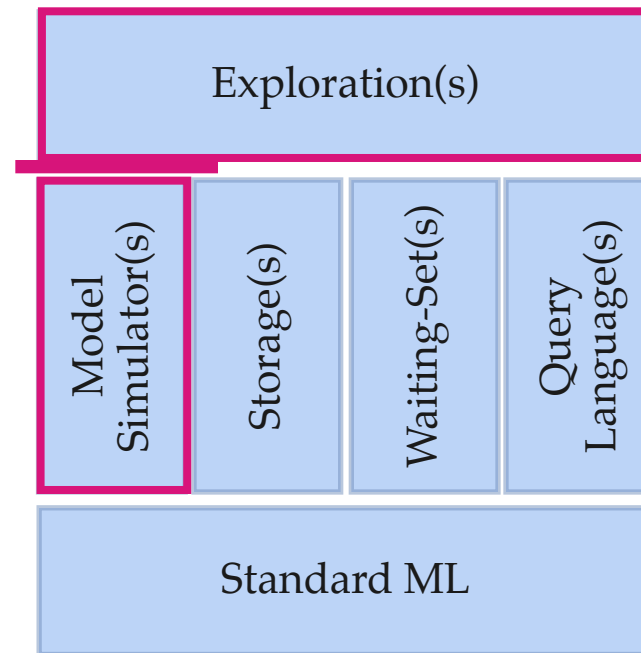
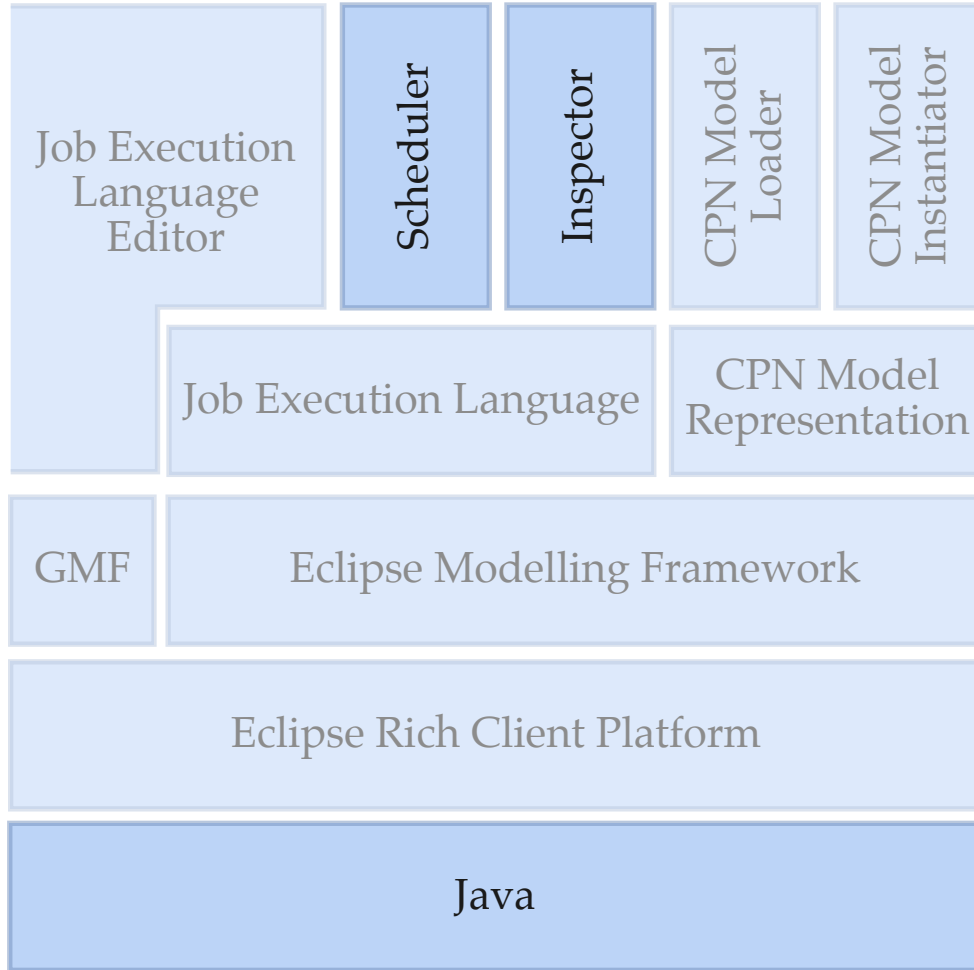


UNIVERSITY OF AARHUS

Department of Computer Science

Michael Westergaard

# Architecture



UNIVERSITY OF AARHUS

Department of Computer Science

Michael Westergaard

# Simple State Space Exploration

```
WaitingSet.enqueue(ModelSimulator.getInitialStates())
Storage.addAll(ModelSimulator.getInitialStates())
while !WaitingSet.empty() do
    (s, e) := WaitingSet.dequeue()
    process(s)
    forall (s', e') in ModelSimulator.nextStates(s, e) do
        if (!Storage.contains(s') then
            Storage.add(s')
            WaitingSet.add(s', e')
        endif
    endfor
endwhile
```



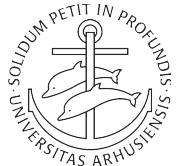
# Simple State Space Exploration

```
WaitingSet.enqueue(ModelSimulator.getInitialStates())
Storage.addAll(ModelSimulator.getInitialStates())
while !WaitingSet.empty() do
  (s, e) := WaitingSet.dequeue()
  process(s)
  for all (s', e') in ModelSimulator.nextStates(s, e) do
    if (!Storage.contains(s')) then
      Storage.add(s')
      WaitingSet.add(s', e')
    endif
  endfor
endwhile
```

Seems nice and clean

— easy to write

easy-to-read algorithms?



# Simple State Space Exploration

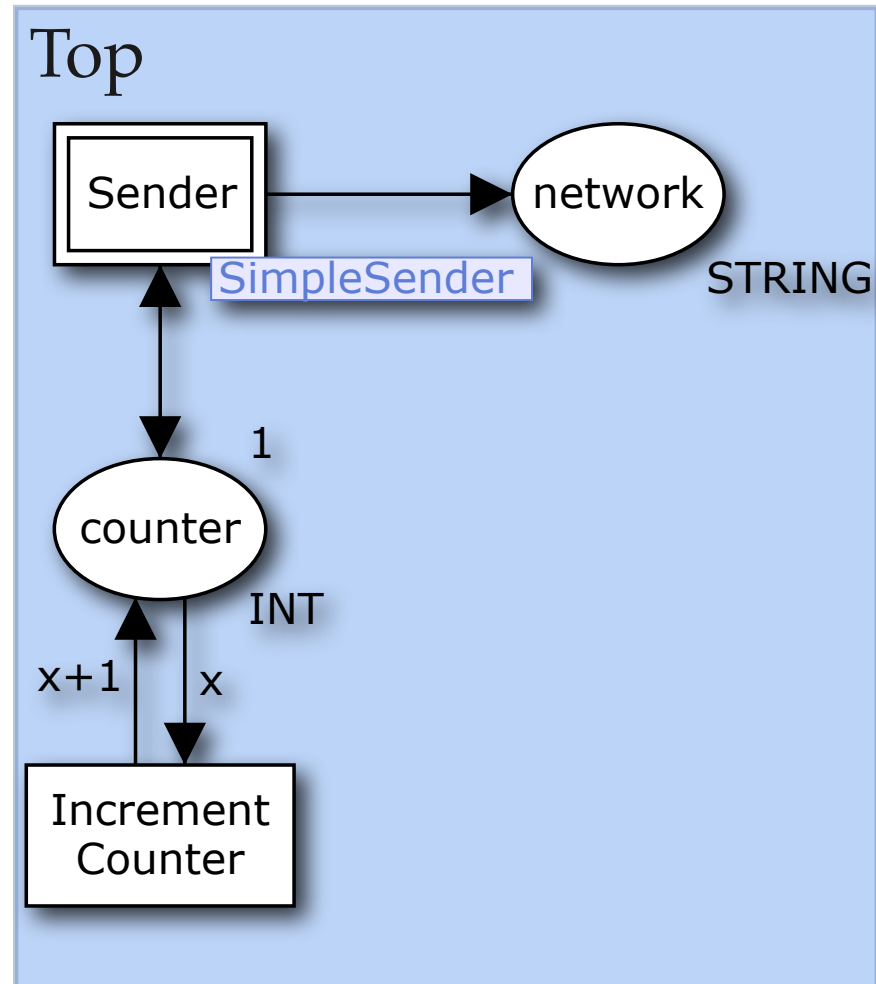
```
WaitingSet.enqueue(ModelSimulator.getInitialStates())
Storage.addAll(ModelSimulator.getInitialStates())
while !WaitingSet.empty() do
  (s, e) := WaitingSet.dequeue()
  process(s)
  for all (s', e') in ModelSimulator.nextStates(s, e) do
    if (!Storage.contains(s')) then
      Storage.add(s')
      WaitingSet.add(s', e')
    endif
  endfor
endwhile
```

Seems nice and clean  
— easy to write  
easy-to-read algorithms?

Unfortunately no...



# Example: Dumb Protocol





# CPN Model Simulator: Pseudo-code

```
MultiSet<String> placeid737;  
MultiSet<Integer> placeid3781;  
MultiSet<Pair<String, Integer>> placeid1782;  
  
void executetransition8357(int v1, String v2) {...}  
void executetransition1783(int v1) {...}  
  
List<Map<String, String>> enabledtransid8357() {...}  
List<Map<String, String>> enabledtransid1783() {...}
```



# CPN Model Simulator: States

- **The state of the system is distributed over multiple global mutable structures**
  - Makes it easy to exploit locality during simulation
  - It is cumbersome to inspect the state of the entire system
  - It is difficult/impossible to work with more than one state of the system



# CPN Model Simulator: Events

- **Enabling and execution of events is distributed over multiple functions**
  - Makes it easy to exploit locality during enabling calculation
  - Makes it cumbersome to get all enabled transition modes



# Simple Exploration

```
WaitingSet.enqueue(ModelSimulator.getInitialStates())
Storage.addAll(ModelSimulator.getInitialStates())
while !WaitingSet.empty() do
    (s, e) := WaitingSet.dequeue()
    process(s)
    forall (s', e') in ModelSimulator.nextStates(s, e) do
        if (!Storage.contains(s') then
            Storage.add(s')
            WaitingSet.add(s', e')
        endif
    endfor
endwhile
```



# Simple Exploration

```
WaitingSet.enqueue(ModelSimulator.getInitialStates())
Storage.addAll(ModelSimulator.getInitialStates())
while !WaitingSet.empty() do
    (s, e) := WaitingSet.dequeue()
    process(s)
    forall (s', e') in ModelSimulator.nextStates(s, e) do
        if (!Storage.contains(s') then
            Storage.add(s')
            WaitingSet.add(s', e')
        endif
    endfor
endwhile
```



# Desired Interface

**signature** MODEL\_SIMULATOR = **sig**

**type** state

**type** event

**val** getInitialStates: *unit* -> (state \* event *list*) *list*

**val** nextStates :

state \* event -> (state \* event *list*) *list*

**end**



# Desired Interface

Allows us to  
handle non-  
deterministic  
formalisms

**signature** MODEL\_SIMULATOR

**type** state

**type** event

**val** getInitialStates: *unit* -> (state \* event *list*) *list*

**val** nextStates :

state \* event -> (state \* event *list*) *list*

**end**



# Desired Interface

**signature** MODEL\_SIMULATOR = **sig**

**type** state

**type** event

**val** getInitialStates: *unit* -> (state \* event *list*) *list*

**val** nextStates :

state \* event -> (state \* event *list*) *list*

**end**



# Desired Interface

Allows more efficient calculation of successors (at least for CP-nets, due to locality)

**signature**

**type** state

**type** event

**val** getInitialStates: *unit* -> (state \* event *list*) *list*

**val** nextStates :

state \* event -> (state \* event *list*) *list*

**end**



# Desired Interface

**signature** MODEL\_SIMULATOR = **sig**

**type** state

**type** event

**val** getInitialStates: *unit* -> (state \* event *list*) *list*

**val** nextStates :

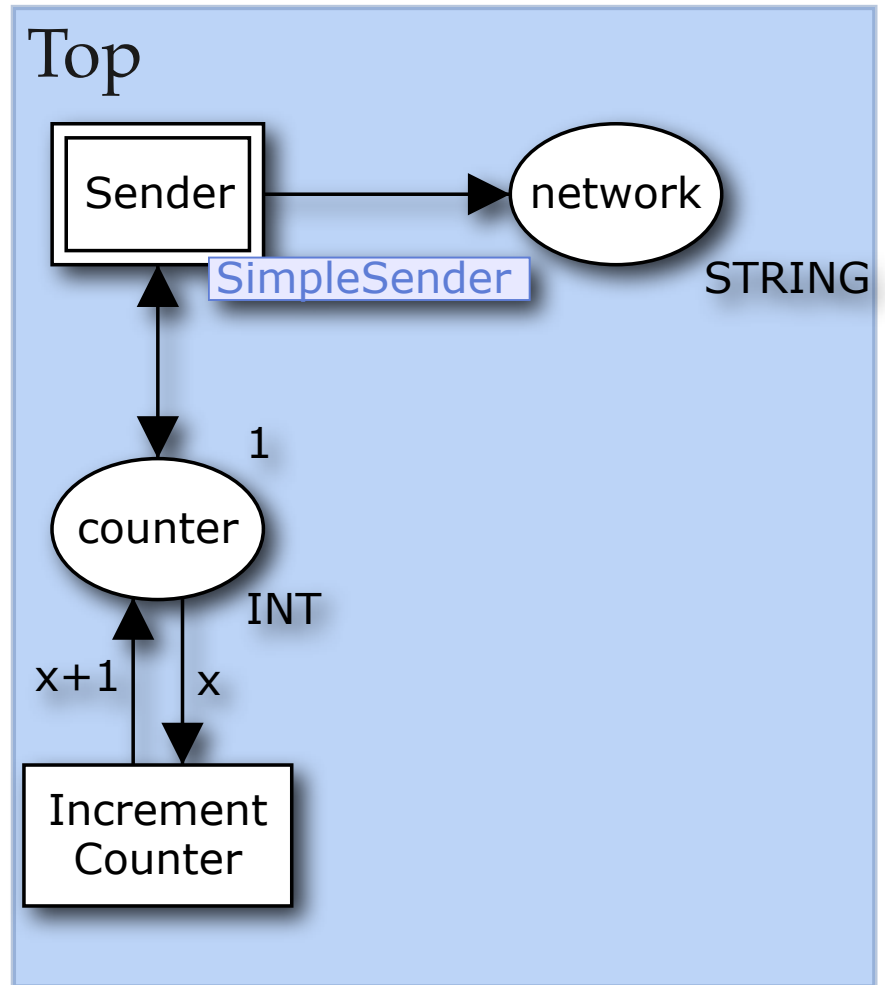
state \* event -> (state \* event *list*) *list*

**end**



# A Simpler State Concept

```
type SimpleSender = {  
  data: INTxSTRING ms  
}  
type Top = {  
  network: STRING ms,  
  counter: INT ms,  
  Sender: SimpleSender  
}  
type state = {  
  Top: Top  
}
```



# A Simpler State Concept

```
type SimpleSender = {  
  data: INTxSTRING ms  
}  
type Top = {  
  network: STRING ms,  
  counter: INT ms,  
  Sender: SimpleSender  
}  
type state = {  
  Top: Top  
}
```

- **Immutable**
  - Can work with multiple states
- **Hierarchical**
  - Can work with the entire state at once
  - Can exploit locality
- **Uses names rather than internal IDs**



# A (Slightly) Simpler Event Concept

```
datatype event = FAKE
```

```
| Top'Increment_Counter of { x: INT }
```

```
| SimpleSender'Send_Packet of { x: INT, data : STRING }
```

- **Unifies all events as a single type**
  - We can write  
getInitialStates: unit -> (state \* event *list*) *list*
- **Not hierarchical because constructors of data-types in SML are unscoped**



# Using the Simpler Types

- **The new state and event types depend heavily on the CPN model**
- **Generic code to translate between the two formats will be slow, as it has to look in a lot of hash-tables**
- **Instead we generate model-specific code for translating between internal simulator representation and our state/event representations**



# getCurrentState

```
fun getCurrentState() =  
{  
    Top = {  
        counter = placeid3781.get(),  
        network = placeid737.get(),  
        Sender = {  
            data = placeid1783.get()  
        }  
    }  
}
```



# getNextStates

```
fun getNextStates(state, event) =  
let  
  val _ = setCurrentState state  
  fun execute' (SimpleSender'Send_Packet { x, data }) =  
    executetransid8357(x, data)  
    | execute' (Top'Increment_Counter { x }) =  
      executetransid1783(x)  
  val _ = execute' event  
in  
  [(getCurrentState(), getEnabledTransitions())]  
end
```



# Model-specific Code

- **We have seen how we can generate model-specific code for getting the state and executing transitions**
- **We may also want to generate other kinds of model-specific code**
  - toString functions for states and events
  - hash functions
  - state and event orders
  - serializer functions (distributed and external algorithms)
  - super-efficient storages



# NetCapture

- **Rather than forcing all code-generators to traverse internal data-structures of CPN Tools, we write a single traversal, which generates a description of the model**
- **This minimises the dependencies on the CPN Tools simulator**
  - 916 lines of code are completely independent
  - 392 lines of code depend on CPN Tools' simulator
  - 57 lines of code depends on the model
- **Switch to state space tool takes < 1s**

